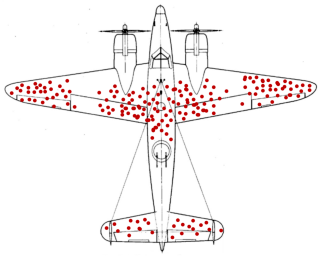
Methodology
for data pre-processing

Machine Learning 2020
mlvu.github.io

To motivate today's lecture, let's look at a famous historical case of operations research. In the second World War, the allies executed many bombing runs, and often, their planes came back like this.

Investigators made a tally of the most common points on the plane they were seeing damage, to investigate where they should reinforce their planes. The initial instinct was to reinforce those places that registered the most hits.

don't take your data at face value



By McGeddon - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=53081927

3

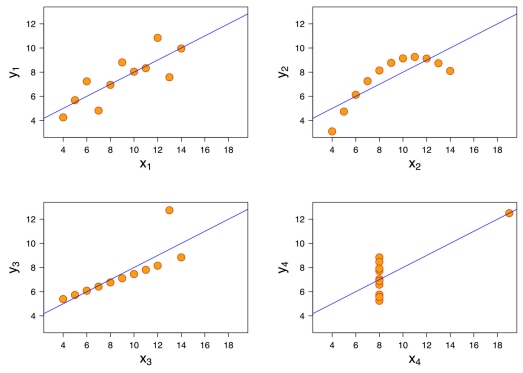
However, it was soon pointed out (by a man called Abraham Wald) that this ignores a crucial aspect of the *source* of the data. They weren't tallying where planes were most likely to be hit, they were tallying where planes were most likely to be hit *and come back*.

The places where they weren't seeing *any* hits were exactly the places that should be reinforced, since the planes that were hit there didn't make it back.

This specific effect is called survivorship bias (and it's worth keeping in mind), but the more general lesson for today, is that you should not take your data at face value.

Don't just load your data into an ML model and check the predictive performance: consider what you're ultimately trying to achieve, and consider where your data came from.

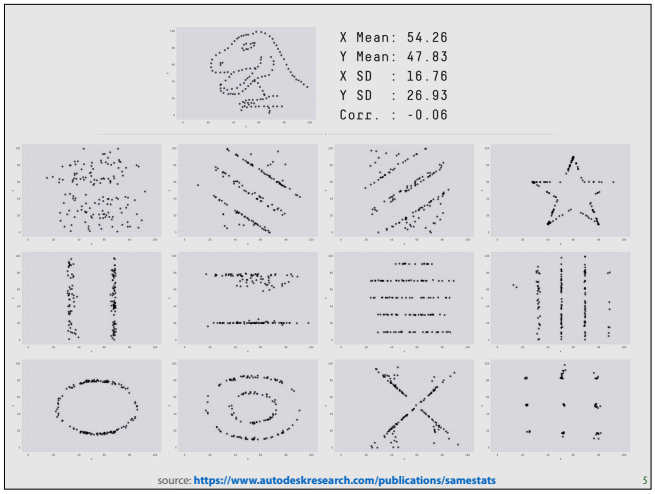
look at your data



source: By Anscombe.svg: Schutz(label using subscripts): Avenue - Anscombe.svg, CC BY-SA 3.0, **https://commons.wikimedia.org/w/index.php?curid=9838454**

4

One important aspect of not taking the data at face value is to *look* at it. These are four datasets (called Anscombe's quartet) with the same correlation, and the same means and variances in all directions. However, they are fundamentally *very* different.



Here is a more modern variant: the datasaurus dozen.

Recommended reading: **https://www.autodeskresearch.com/publications/samestats**

---

methodology

part 1:

Cleaning your data
Missing data, outliers, class imbalance

Choosing features
Creating numeric or categorical features, transforming features

part 2:

Normalisation

Dimensionality reduction
Principal Component Analysis

Eigenfaces

6

---

cleaning your data

- Missing data:
  missing labels
  missing values
- Outliers
- Class imbalance

7

---

missing values

income	status	unemployed
32000	married	true
	single	false
89000		true
34000	divorced	false
54000	married	true
		false
21000		true
25000	single	true

8

Let's start with missing data. Quite often, your data will look like this. You will need to do something about those gaps, before any machine learning algorithm will accept this data.

missing *labels*

income	status	unemployed
32000	married	true
2000	single	
89000	single	true
34000	divorced	false
54000	married	
34000	married	false
21000	divorced	
25000	single	true

Let's start with missing data. Quite often, your data will look like this. You will need to do something about those gaps, before any machine learning algorithm will accept this data.

simple solutions

Remove the feature

Remove the instances

- are the data missing uniformly?

The simplest way to get rid of missing data is to just remove the feature(s) for which there are values missing. If you're lucky, the feature is not important anyway.

You can also remove the instances (i.e. the rows) with missing data. Here you have to be careful. If the data was not corrupted uniformly, removing rows with missing values will change your data distribution.

You might have data gathered by volunteers. If only one volunteer had a hardware problem, then only their data will contain missing values.

Another reason for unequally distributed missing data is if people refuse to answer certain questions. For instance, if only rich people refuse to answer questions, removing these instances will remove lots of rich people from your data and

Think about the REAL-WORLD use case.

Whenever you have questions about how to approach something like this, it's best to think about the real world setting where you might apply you trained model. Can you expect missing data there too, or will that data be clean already?

Examples of production systems that should expect missing data are situations where data comes from a form with optional values or situations where data is merged from different sources (online forms and phone surveys).

will you get missing values in production?

YES:

Keep them in the test set, and make a model that can deal with them them.

NO:

Endeavour to get a test set without missing values, and test different methods for completing the data in the training set only.

By **production**, we mean whatever setting you will deploy your finished model. This could be an actual software production environment, or just script that outputs business predictions.

If you can reasonably assume that the values are missing uniformly, then you can just sample instances without missing values for your test set. Otherwise, you'll have to model the process that corrupted your data. That's outside the scope of this lecture, but one simple trick is to turn "missing" into another category, or to reserve special numeric value (like -1).

How can you tell if data is missing uniformly? There's no surefire way, but usually you can get a good idea by plotting a histogram of how much data is missing against some other feature. For instance if the number of instance with missing features against income is very different from the regular histogram over income, you can assume that your data was not

corrupted uniformly.

imputation: guess the missing values

categorical data: use the mode

numerical data: use the mean

make the feature a target value and train a model
kNN, linear regression, etc.

13

In your training set, you can try to manipulate the data to help your model (don't do this in the test set, though, that's cheating). A simple start is to just fill in a mode or a mean.

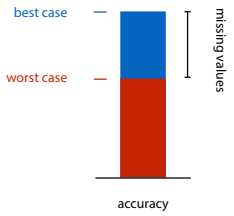
A more involved way is to predict the missing value from the other features. You just turn the feature column in to a target column and train a classifier for categoric features, and a regression model for numeric features.

So long as you have a clean test (and validation) set, you can experiment with any method you like.

missing *labels*

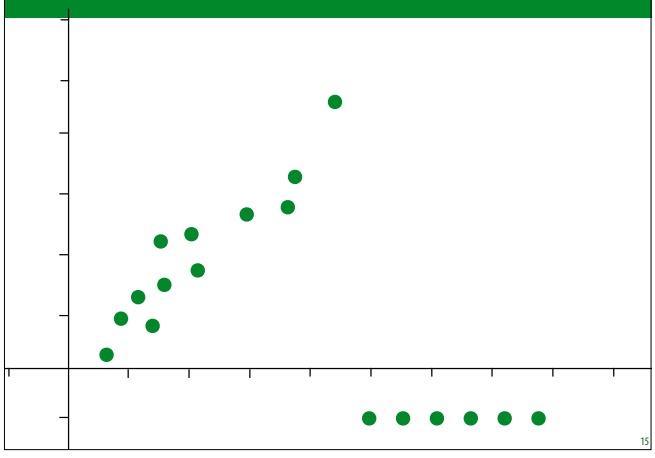
training set
- train only on labeled data
- impute the missing labels
- many missing labels: semi-supervised learning

test set
- don't impute, don't ignore!
- report your uncertainty



14

unnatural outliers



15

Outliers come in different shapes and sizes. Here, the six dots to the right are so oddly, mechanically aligned that we are probably looking at some measurement error (perhaps someone using the value -1 for missing data). We can remove these, or interpret them as missing data, and use the approaches just discussed.

natural outliers

income

16

Here however, the "outlier" is very much part of the distribution. If we fit a normal distribution to this data, the outlier would ruin our fit, but that's because the data isn't normally distributed. Here we should leave the outlier in, and adapt our model.

17

If our instance are image of faces, the image on the left is an extreme of our data distribution. It looks odd, but it can be very helpful in fitting a model. The right is clearly corrupted data, that we may want to clean.

However, remember the real-world use-case: if we can expect corrupted data in production as well, then we should either train the model to deal with it, or clean it automatically in production as well. It may be easier just to leave the outliers in, if they don't cause too much trouble.

---

outliers

Are they mistakes?

- Yes: deal with them.
- No: leave them be. Check your model for strong assumptions of normality.

Can we expect them in production?

- Yes: Make sure the model can deal with them.
- No: Remove. Get a test set that represents the production situation.

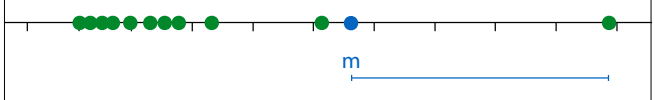
18

If you have very extreme values that are not mistakes (like Bill Gates earlier), your data is probably not normally distributed. If you use a model which assumes normally distributed data (like linear regression), it will be very sensitive to these kinds of "outliers". It may be a good idea to remove this assumption from your model (or replace it by an assumption of a heavy-tailed distribution).

Note that you have to know your model really well to know if there are assumptions of normality. For instance anything that uses squared errors essentially has an assumption of normality.

---

fitting a central value



$$\arg\min_{m} \sum_{i} (m - x_i)^2$$

19

To illustrate: let's learn which single value best represents our data. We choose a value m, compute the distance to all our data points (the residuals) and try to minimise their squares. This is a one dimensional version of the linear regression. The only assumption we've made is that of **squared errors**.

$$\arg\min_m \sum_i (m - x_i)^2$$

$$\sum_i \frac{\partial (m - x_i)^2}{\partial m} = 0$$

$$\sum_i \frac{\partial (m - x_i)^2}{\partial m - x_i} \frac{\partial m - x_i}{\partial m} = 0$$

$$2 \sum_i m - x_i = 0$$

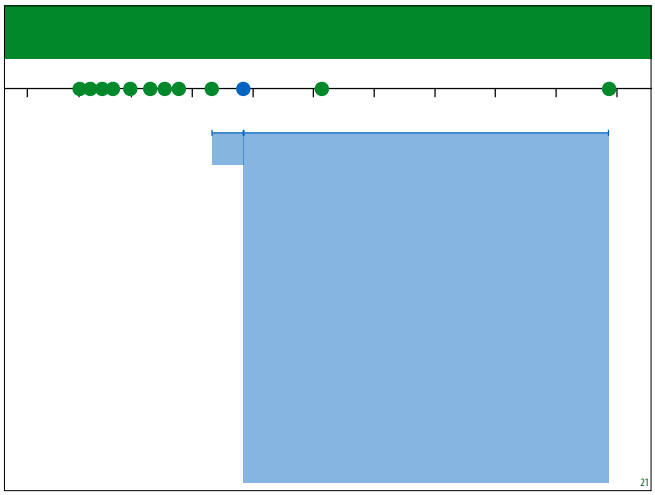
$$nm - \sum_i x_i = 0$$

$$m = \frac{\sum_i x_i}{n}$$

20

We take the derivative of the objective function and set it equal to zero (no gradient decent needed here).

What we find is that the optimum is the mean. The assumption of squared errors leads directly to the use of the mean as a representative example.

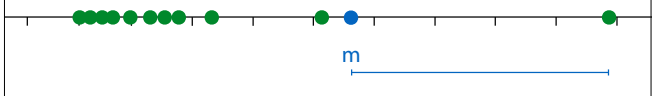


21

Bill gates is not pulling on the mean a million times as much as the person next to the mean, he's pulling 10^{12} times as much.

Hence the joke: a billionaire walks into a homeless shelter and says "What a bunch of freeloaders, the average wealth in this place is more than a million!"

mean absolute error



$$\arg\min_m \sum_i |m - x_i|$$

22

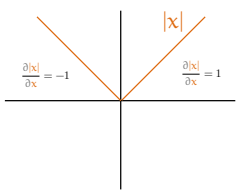
To get rid of the normality assumption, we can use the mean absolute error instead. We take the residuals, but we sum their absolute value instead of their squared value. Which is the most representative value that minimises that error?

$$\arg\min_m \sum_i |m - x_i|$$

$$\sum_i \frac{\partial |m - x_i|}{\partial m} = 0$$

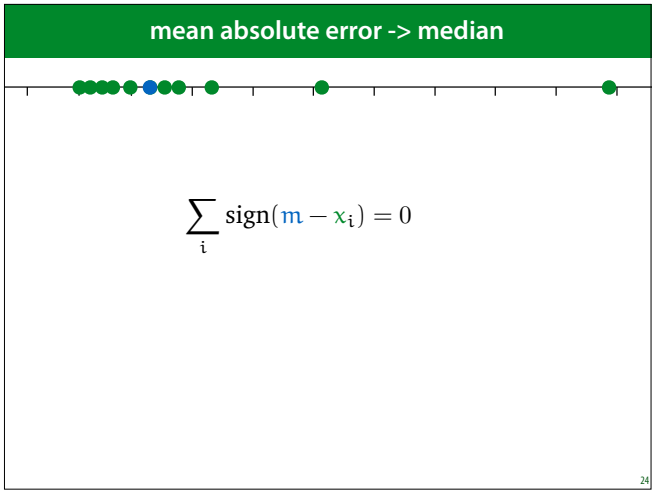
$$\sum_i \frac{\partial |m - x_i|}{\partial m - x_i} \frac{\partial m - x_i}{\partial m} = 0$$

$$\sum_i \text{sign}(m - x_i) = 0$$

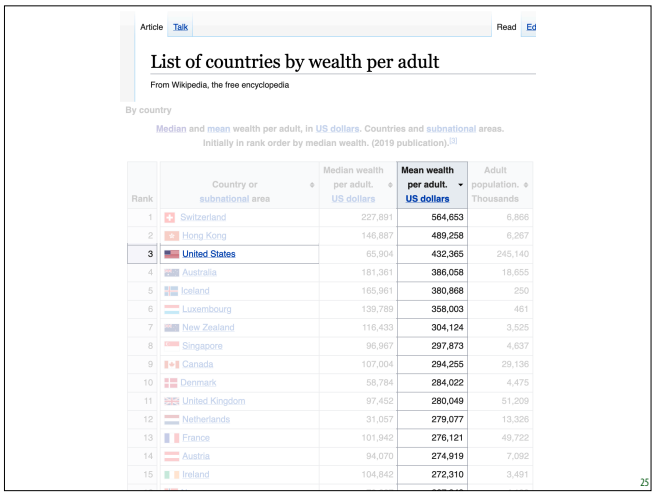


23

To work this out, we need to know the derivative of the absolute function. This function is the identity if the argument is positive (so its derivative is 1) and the negative identity if the argument is negative

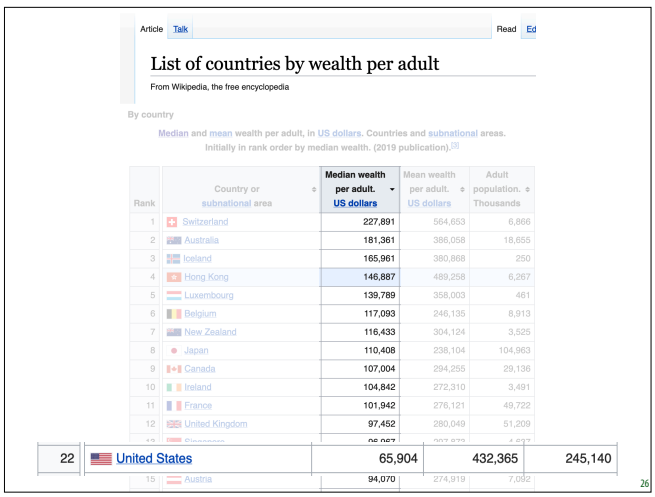


The value of m for which the sum of the signs is zero, is the value that has as many data points to the left as it does to the right. That is: the median.



This is very crucial to the correct interpretation of data. You might hear someone say something like “there’s no poverty in the US, it’s the third richest country in the world by average personal wealth”.

Wikipedia allows us to check this quickly, but it also allows us to sort the same list by *median* wealth.



And here we see that the US suddenly drops to 22nd place.

The Netherlands drops from 12th to 34, incidentally. So there’s plenty of income inequality over here as well.

Here’s an example of this fallacy in the wild. In 2019, there was a big national discussion in the US about unionisation in the game industry. Here, one of the heads of Take-Two suggests that because the *average* yearly salary has six figures, unions are unlikely.

Whether rich people can benefit from unions is a question for a different series of lectures, but the fact that the averages wages are high, most likely just means that there is a small number of very rich people in the industry. We’d need to know the *median* to be sure.

models that can deal with outliers

Beware of squared errors (MSE) week 3

They are only justified if your values have well-determined *scale*

Model noise with a heavy-tailed distribution

Choose the median over the mean.

The proof is in the pudding.

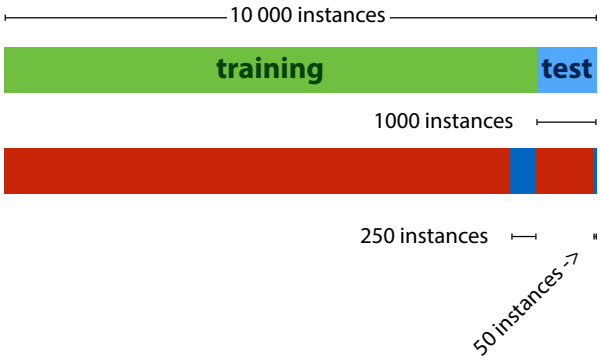
The performance on the test/validation set will be the deciding factor.

28

MSE is based on an assumption of normally distributed randomness. Data with big outliers (like income) violates this assumption.

It pays to be aware of this, but even if your model assumptions are violated, MSE may still produce the best results practically.

class imbalance



29

In the last lecture, we saw that class imbalance can be a big problem. We know what we can do to help our analysis of imbalanced problems, but how do we actually improve training?

class imbalance

Use a big test set.

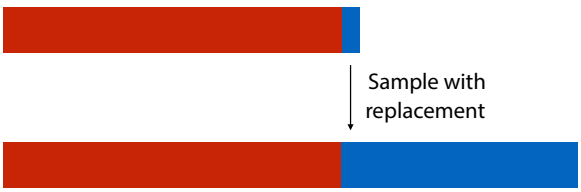
Don't rely on accuracy. Try ROC plots, precision-recall plots, AUC, etc. Look at the confusion matrix.

Resample your training data.

Use data augmentation for the minority class

30

oversampling



31

The most common approach is to oversample your minority class by sampling with replacements. The advantage is that this leads to more data. The disadvantage is that you end up with duplicates in you dataset. This may increase the likelihood of overfitting.

undersampling

You can also under sample your majority class. This doesn't lead to duplicates, but it does mean you're throwing away data.

If you have use an algorithm that makes multiple passes over the dataset (like stochastic gradient descent, explained next week) it can help to resample the dataset fresh for every pass.

data augmentation

SMOTE: add midpoints between nearby minority class data points.

Domain specific: rotate or translate images in the minority class. Add Gaussian noise.

Remember: only on the training data. Keep the test data as is.

more information: https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets

A more sophisticated approach is to oversample the minority class with new data derived from the existing data.

SMOTE is a good example: it finds small clusters of points in the minority class, and generate their mean as a new minority class point. This way, the new point is not a duplicate of any existing point, but it is still in a region that contains a lot of points in the minority class.

We don't have time to go into this deeply. If you run into this problem in your project, click the link for a better explanation.

getting features

phone nr	income	status	unemployed	birthdate	age
0646785910	32000	married	true	4-5-78	41
0207859461	45000	single	false	3-6-00	19
0218945958	89000	married	true	4-7-91	28
0645789384	34000	divorced	false	3-11-94	25
0652438904	54000	married	true	21-3-95	24
0309897969	36000	single	false	4-12-46	73
0159874645	21000	single	true	13-8-52	67
0256789456	25000	single	true	16-8-79	40

Even if your data comes in a table, that doesn't necessary mean that every column can be used as a feature right away (or that this would be a good approach).

from: date, phone number, images, status, text, category, tags, etc…

to: numeric, categoric, both.

Some algorithms (like linear models or kNN) work only on numeric features. Some work only on categorical features, and some can accept a mix of both (like decision trees). Translating your raw data into features is more an art than a science, and the ultimate test is the test set performance. But let's look at a few examples, to get a general sense of the way of thinking.

age

age
41
19
28
25
24
73
67

to numeric: From integer to real-valued. Not usually an issue.

to categoric: Group data into bins? E.g. above or below the median.

- Information loss is unavoidable.

36

Age is integer valued, while numeric features are usually real-valued. In this case, we can just interpret the age as a real-valued number, and most algorithms won't be affected.

If our algorithm only accepts categoric features, we'll have to group the data into bins. For instance, you can turn the data into a two-category feature with the categories "below the median" and "above the median".

We'll lose information this way, which is unavoidable, but if you have a classifier that only consumes categorical features, and works really well on your data, it may be worth it.

phone number

phone nr
0646785910
0207859461
0218945958
0645789384
0652438904
0309897969
0159874645

to numeric: From integer (?) to real-valued. Highly problematic.

to categoric: area codes, cell phone vs. landline

37

We can represent phone numbers as integers too, so you might think the translation to numeric is fine. But here it makes no sense at all. Translating to a real valued feature would impose an *ordering* on the phone numbers that would be totally meaningless. My phone number may represent a higher number than yours, but that has no bearing on any possible target value.

What *is* potentially useful information, is the area code. This tells us where a person lives, which gives an indication of their age, their political leanings, their income, etc. Wether or not the phone number is for a mobile or a landline may also be useful. But these are **categorical features**.

categoric to numeric

integer coding:

genre	genre
sci-fi	1
romance	2
comedy	4
thriller	3
thriller	3
romance	2
romance	2
sci-fi	1
thriller	3
comedy	4

one-hot coding:

scifi	romance	comedy	thriller
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
0	0	0	1
0	1	0	0
0	1	0	0
1	0	0	0
0	0	0	1
0	0	1	0

aka 1-of-N coding

38

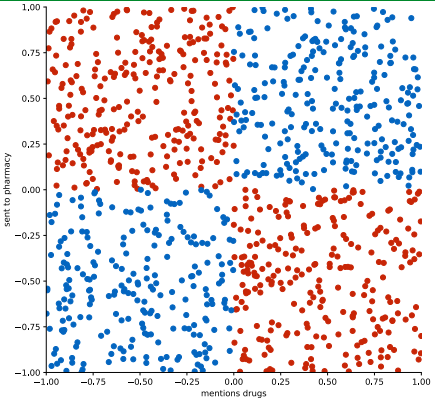
So what if our model only accepts numerical features? This is very common: most modern machine learning algorithms are purely numeric. How do we feed it categorical data? Here are two approaches.

Integer coding gives us the same problem we had earlier. We are imposing a false ordering on unordered data.

One-hot coding avoids this issue, by turning *one* categorical feature into *several* numeric features. Per genre we can say whether it applies to the instance or not.

In general, the one-hot coding approach is preferable.

expanding features



39

Once we've turned all our features into data that our model can handle, we can still manipulate the data further, to improve performance.

How to get the useful information from your data into your classifier depends entirely on what your classifier can handle. The linear classifier is a good example. It's quite limited in what kinds of relations it can represent. Essentially, each feature can only influence the classification boundary in a simple way. It can push it up or down, but it can't let its influence depend on the values of the other features. Here is a (slightly contrived) example of when that might be necessary.

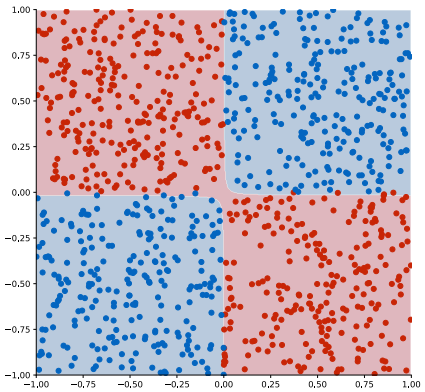
Imagine classifying spam emails on two features: to what extent the email mentions drugs, and to what extent the email is sent to a pharmaceutical company. This problem is completely impossible for a linear classifier.

cross product

d	p	d * p
0.75	0.98	0.74
-0.66	-0.32	0.21
-0.45	0.84	-0.38
0.93	0.78	0.72
-0.42	0.24	-0.10
-0.02	0.43	-0.01
-0.74	0.58	-0.43
-0.41	-0.41	0.17
0.59	0.72	0.42

40

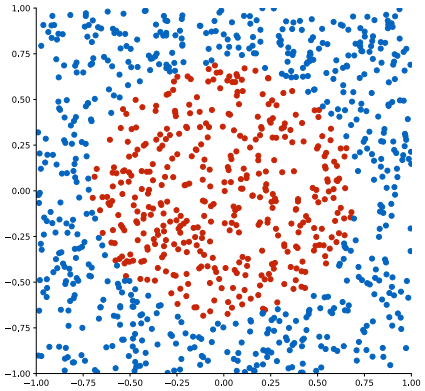
We can switch to a more powerful model, but we can also add power to the linear classifier by **adding extra features derived from the existing features**. Here, we've added the cross-product of d and p. (Note the XOR relationship of the signs: two negatives or two positives both make positive, a negative and a positive make a negative). Now the classifier can separate the classes perfectly by just looking at whether or not the third column is positive or negative.



41

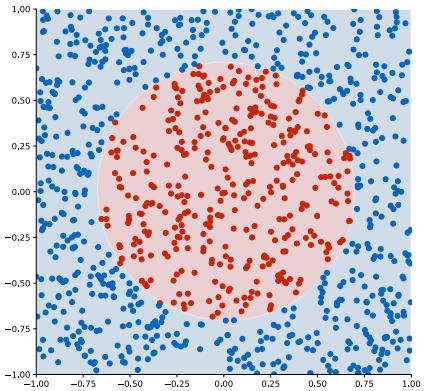
This is a linear classier that operates in a 3D space. But since the third dimension is derived from the other two, we can colour our original 3D space with the classifications. Projected down to 2D, the classifier solves our XOR problem perfectly.

You can try this yourself at **playground.tensorflow.org**.

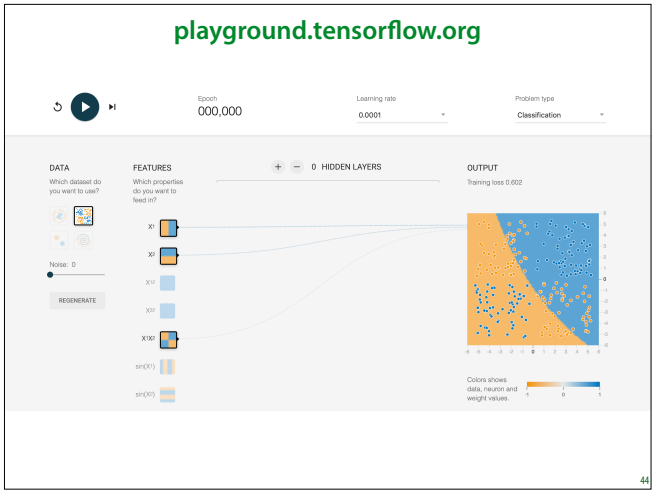


42

One more example. Here we color points red if the distance to the origin is less than 0.7. Again, this dataset is not at all linearly separable. Using Pythagoras, however, we can express how the classes are decided: if $x_1^2 + x_2^2 < 0.7^2$ then we classify as red, otherwise as blue. This is a linear decision boundary for the features x_1^2 and x_2^2.

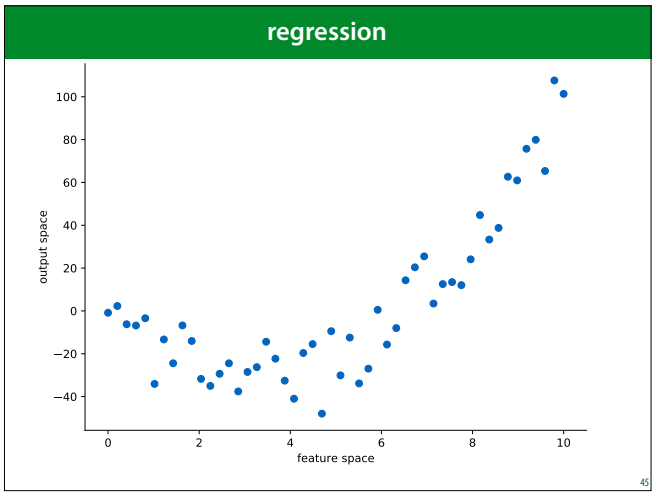


43

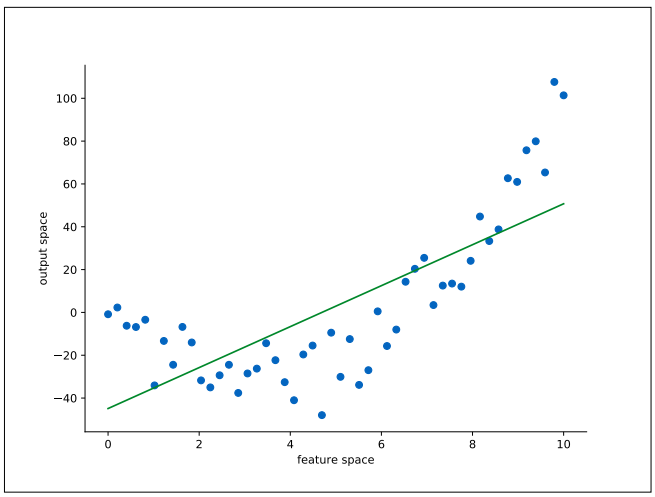


You can test this idea in the tensorflow playground. Just turn on some of the extra features in the `features' column.

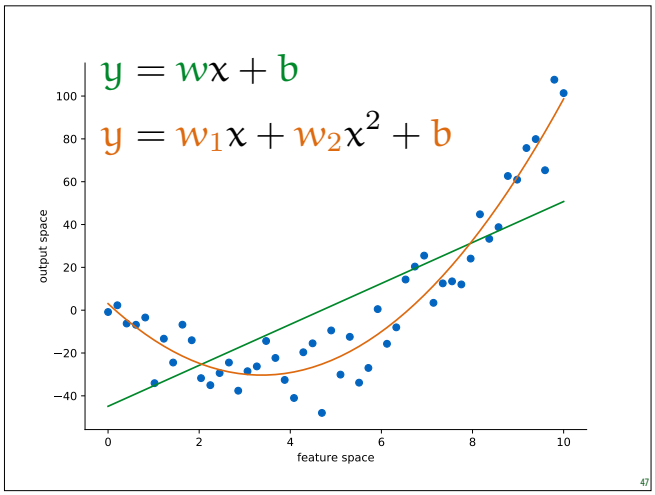
https://bit.ly/2SI3Krp



We can do the same thing with regression. Here, we have a very non-linear relation.



A purely linear classifier does a terrible job.



We *can* fit a parabola through the data perfectly. We can see this as a more powerful model, but we can also see this as a 2D linear regression problem, where the second feature (x^2) is derived from the first.

This is relevant because linear models are extremely simple to fit. By adding derived features we can have our cake and eat it too. A *simple* model that we can fit quickly and accurately, and a *powerful* model that can fit many nonlinear aspects of the data.

If we don't have any intuition for which extra features might be worth adding, **we can just add all cross products**. Other functions like the sine or the logarithm may also help a lot.

adding features

Can make a weak classifier (especiall a linear one) stronger.

Any function on one or more of the existing features can work.

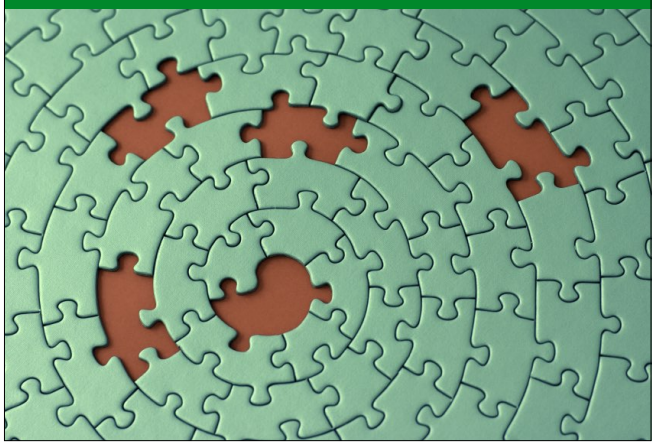
The model stays *convex*: easy to solve, optimal solution guaranteed.

Common choice: all 2-way cross products, all 3-way cross products, etc.

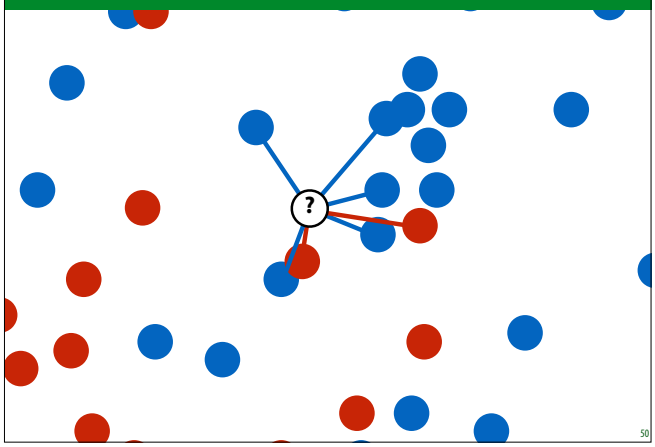
$x, y, z \rightarrow x, y, z, xy, zy, xy, x^2, y^2, z^2, xyz, x^2y, \ldots, x^3, \ldots$

48

break

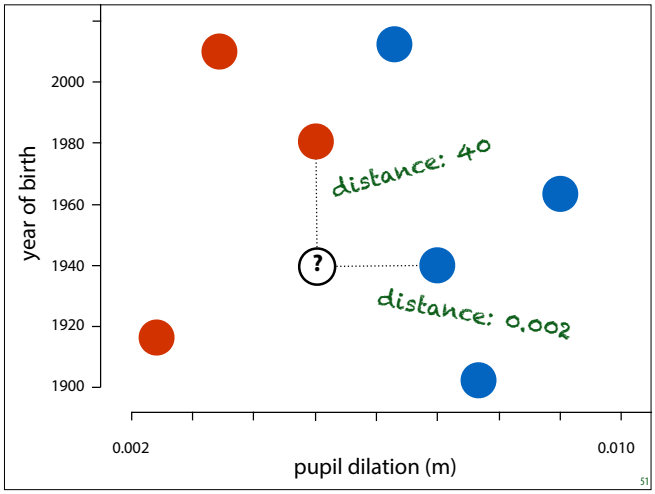


normalization



50

It can help to make sure that all numeric features have broadly the same maximum and minimum. In other words, that they are normalized. To see why, et's go back to the kNN classifier.



51

Imagine we are using a 1-NN classifier (i.e. it only looks at the nearest example, and copies its class).

In this plot, it looks like the blue and the red dot are the same distance away. Actually, because years are so much bigger than pupils, the blue dot will always be much closer. But this distinction is not meaningful. What we want to look at is how much spread there is *in the data*, and use that as our natural distance.

We do that by normalising our data before feeding it to the model.

creating a uniform scale

normalization
fit to [0,1]

standardization
fit to 1D standard normal distribution

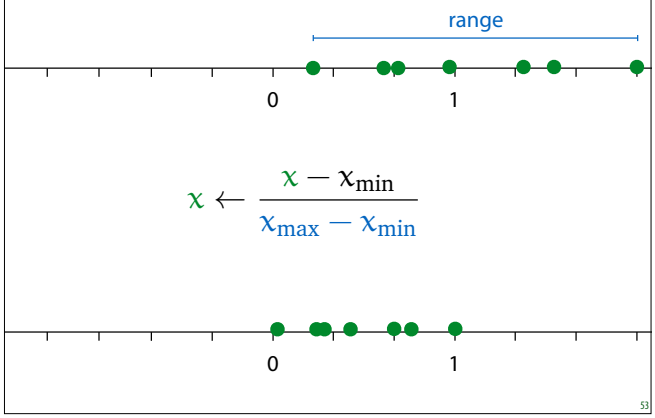
whitening
fit to *multivariate* standard normal distribution

52

These terms are often used interchangeably. We'll stick to these definitions for this course, but in other contexts you should check that they mean what you think they mean.

---

normalization



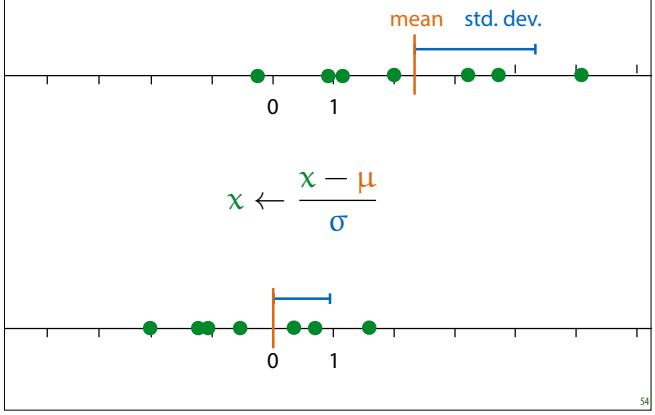
53

Normalisation scales the data linearly so that the smallest point becomes 0 and the largest becomes 1. Note that because x_{min} is negative (in this example), we are actually moving all data to the right, and then rescaling it.

We do this independently for each feature.

---

standardization



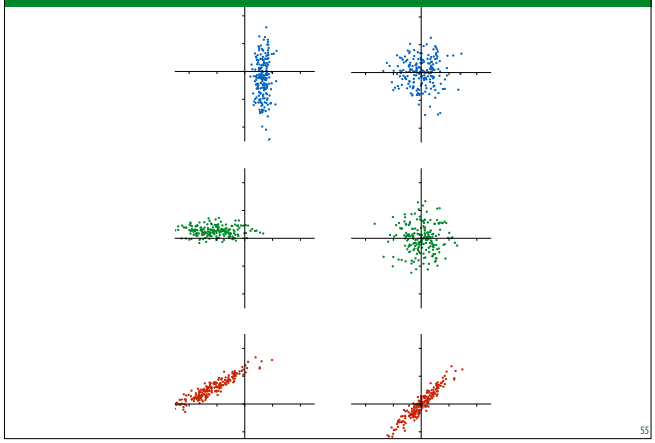
54

Another option is standardization. We rescale the data so that the mean becomes zero, and the standard deviation becomes 1. In essence, we are transforming our data so that it looks like it was sampled from a standard normal distribution (as much as we can with a one dimensional linear transformation).

We can think of the data as being generated from a standard normal distribution, followed by multiplication by sigma, and and adding mu. The result is the distribution of the data. If we then compute the mean and the standard deviation of the data, the formula in the slide is essentially inverting the transformation, recovering the "original" data as sampled from the normal distribution.We will build on this perspective to explain whitening.

---

whitening



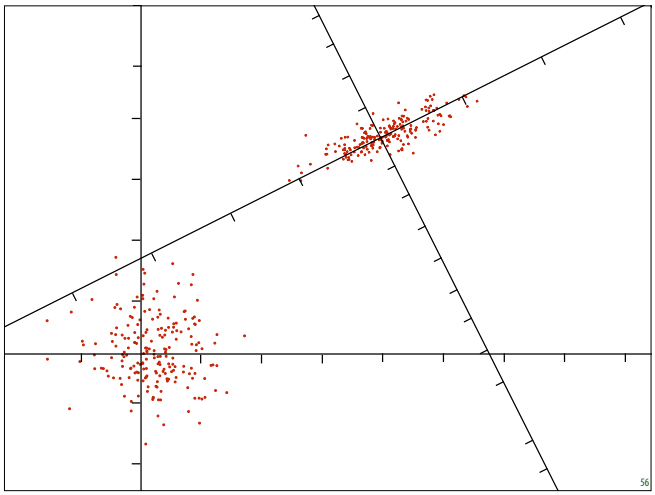
55

Here's what standardization looks like if we apply it to data with two features. If the data is *uncorrelated*, we are reducing it to a nice spherical distribution, centered on the origin, with the same variance in each direction. Exactly what data from a multivariate standard normal distribution looks like.

If, however, our data is correlated (knowing the value of one feature helps us predict the value of the other), we get a different result. This is because we standardize each feature independently, and the features are not independent. Is there a way to achieve the same effect with the correlated data? Can we transform the features somehow so that it looks like they came from a distribution like the one top right?

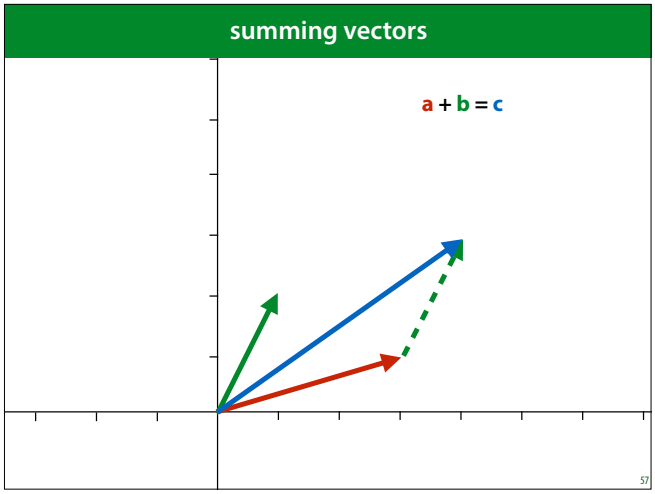
Note that this is not usually necessary in practice. Normalizing or standardising each feature independently is usually fine. especially if the your model is powerful enough

to learn correlations. However, for the rest of the lecture, it's instructive to see how to perform this transformation.

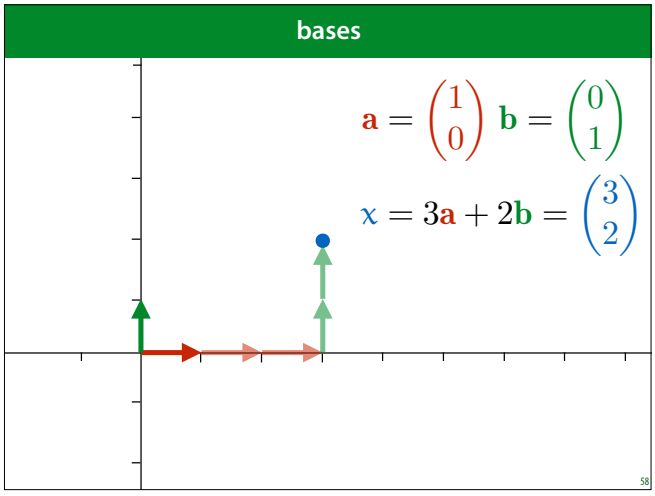


In essence we want to transform the data top right to something that looks like the data bottom left. Or, the same question asked differently, can we express the data in another **coordinate system**, to that in the new coordinate system, the features are not correlated and the variance in the direction of each axis is 1?

In order to show how to do this we need to revise some bits of linear algebra. Specifically, we need to look at **linear bases** (the plural of basis).



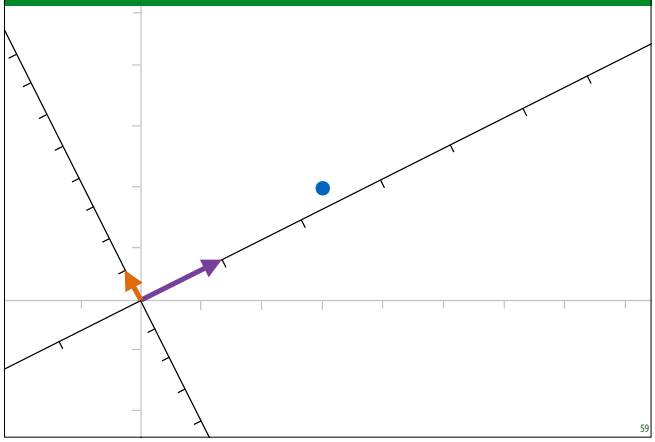
Here's a quick reminder of how summing vectors works.



We can see our modern Cartesian coordinate system as made up entirely of the two vectors (1 0) and (0 1). To describe a point in the place, we just sum a number of copies of these vectors.

Every point in the plane is just a linear combination of these two. A coordinate like (3, 2) means: "sum three copies of b_1 and add them to two copies of b_2."

bases



If we choose different **basis vectors**, we get a different coordinate system to express our data in. But (excepting some rare choice of basis vectors), we can still express all the same points as a number of copies of one vector, plus a number of copies of the other.

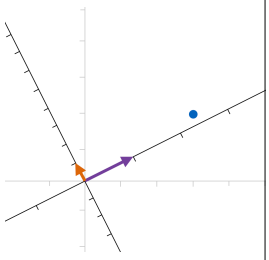
---

bases

$$\mathbf{c} = \begin{pmatrix} 1.26 \\ 0.9 \end{pmatrix} \quad \mathbf{d} = \begin{pmatrix} -0.3 \\ 0.5 \end{pmatrix}$$

$$\mathbf{x}_s = 2.5\mathbf{c} + 0.5\mathbf{d} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

$$\mathbf{x}_b = \begin{pmatrix} 2.5 \\ 0.5 \end{pmatrix}$$



If we know the coordinates $\mathbf{x}_b$ in our non-standard coordinate system, it's easy to find the standard coordinates $\mathbf{x}_s$. We just multiply the first coordinate of x_b with the first basis vector, the second coordinate with the second basis vector and sum the result.

---

rebasing

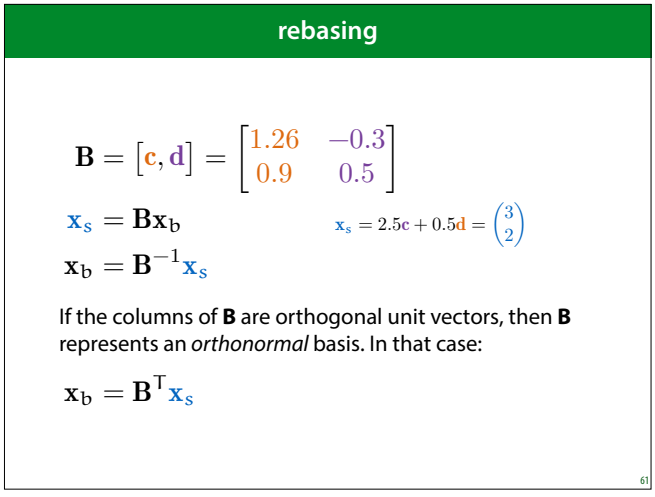
$$\mathbf{B} = [\mathbf{c}, \mathbf{d}] = \begin{bmatrix} 1.26 & -0.3 \\ 0.9 & 0.5 \end{bmatrix}$$

$$\mathbf{x}_s = \mathbf{B}\mathbf{x}_b \qquad \mathbf{x}_s = 2.5\mathbf{c} + 0.5\mathbf{d} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

$$\mathbf{x}_b = \mathbf{B}^{-1}\mathbf{x}_s$$

If the columns of **B** are orthogonal unit vectors, then **B** represents an *orthonormal* basis. In that case:

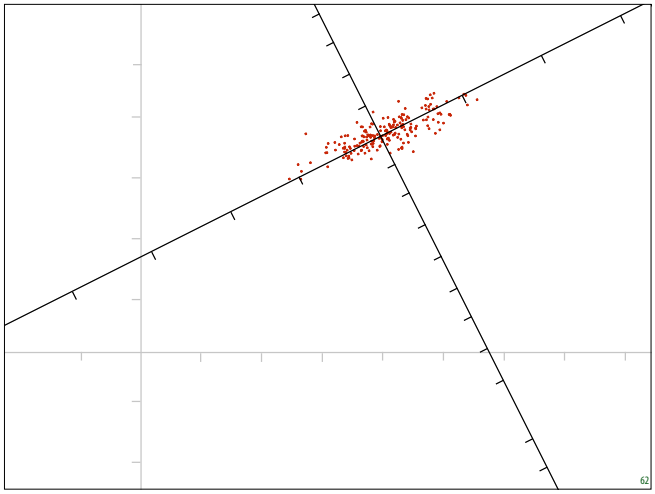
$$\mathbf{x}_b = \mathbf{B}^{\mathsf{T}}\mathbf{x}_s$$



The basis vectors are usually expressed as the columns of a matrix B. That way, transforming a coordinate in basis B to the standard coordidates can be done simply by multiplying by B. It also tells us that to go the other way, to transform a standard coordinate to the basis, you multiply by the inverse of B.

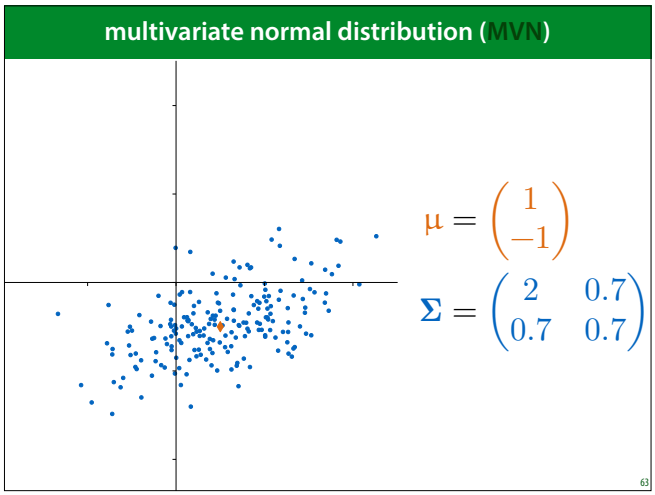
Since inverting a matrix is an expensive and numerically unstable business, it's good to focus (if possible) on **orthonormal bases**. That is, bases for which the basis vectors are **orthogonal** (the angle between any two bases is 90 degrees) and **normal** (all vectors have length 1). In that case the matrix transpose (which is simple to compute without loss of precision) is equal to the matrix inverse, so we can switch back and forth between bases quickly, without losing information.

Here [a,b] represents the matrix created by concatenating the vectors a and b.



We can now re-phrase what we're aiming to do: we want to find a set of new basis vectors so that we can express the data in a coordinate system where the features are not correlated, and the variance is 1 in every direction.

Note that the latter means we can't have an orthonormal basis (the basis vectors can't be one). In week 6 we'll see how we can have our cake and eat it too.

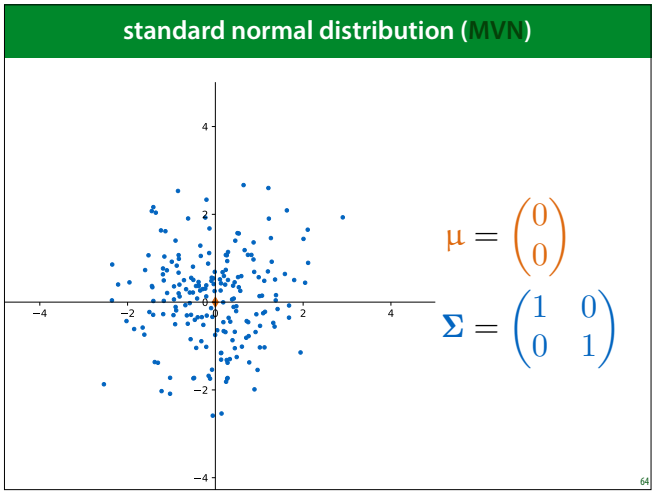


To figure out how to find this basis, we will follow the same principle as we did with standardisation: we sill assume that the data was generated by a multivariate normal distribution (MVN), followed by a translation and a change of basis (with the change of basis causing some features to become correlated). We will attempt to reverse the process by:

- fit a (nonstandard) MVN to the data
- figure out the transformation that transforms the standard MVN to this MVN
- apply the inverse of this transformation.

A multivariate normal distribution is a generalisation of a one-dimensional normal distribution. Its mean is a single point, and its variance is determined by a symmetric matrix called a covariance matrix. The values on the diagonal indicate how much variance there is along each dimension. The off-diagonal elements indicate how much co-variance there is between dimensions.

The *standard* MVN has its mean at the origin and the identity matrix as its covariance matrix (i.e. its features are uncorrelated, and the variance is 1 along every dimension).



To figure out how to find this basis, we will follow the same principle as we did with standardisation: we sill assume that the data was generated by a multivariate normal distribution (MVN), followed by a translation and a change of basis (with the change of basis causing some features to become correlated). We will attempt to reverse the process by:

- fit a (nonstandard) MVN to the data
- figure out the transformation that transforms the standard MVN to this MVN
- apply the inverse of this transformation.

A multivariate normal distribution is a generalisation of a one-dimensional normal distribution. Its mean is a single point, and its variance is determined by a symmetric matrix called a covariance matrix. The values on the diagonal indicate how

much variance there is along each dimension. The off-diagonal elements indicate how much co-variance there is between dimensions.

The *standard* MVN has its mean at the origin and the identity matrix as its covariance matrix (i.e. its features are uncorrelated, and the variance is 1 along every dimension).

---

fitting an MVN to data

$$\mathbf{m} = \frac{1}{n}\sum_i \mathbf{x}_i$$

$$\mathbf{X} = \left[\mathbf{x}_1, \ldots, \mathbf{x}_n\right] - \mathbf{m}$$

$$\mathbf{S} = \frac{1}{n-1}\mathbf{X}\mathbf{X}^T$$

65

The estimators for the **sample mean** and **sample covariance** look like this. Computing these values lets you fit an MVN to your data.

MVNs as transformations

Start with N(**0**, **I**) <- standard normal

- Let X ~ N(**0**, **I**)
- Let Y = **A**X + **t**

then:

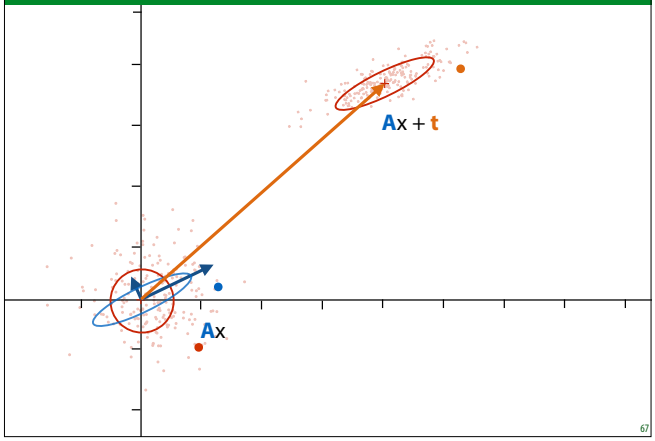
Y ~ N(**t**, **AA**^{T})

66

The transformation is easiest to understand if you think about sampling from the nonstandard MVN.

We can *sample* from an n-dimensional **Standard MVN** (with mean at the origin and the identity matrix I as a covariance matrix) by simply filling a length n vector with values sampled from a one-dimensional standard normal distribution. If we then transform x by multiplying it by some matrix A and adding some vector t, the result is the same as sampling from an MVN with mean **t** and covariance **AA**^{T}.

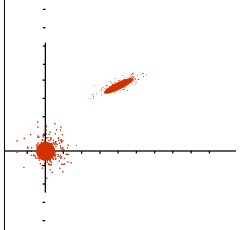
MVNs as transformations



67

We can think of any nonstandard MVN as a linear transformation of the standard MVN.

whitening: invert this transformation



- Compute **S**, **m** from data.
- Find some **A** such that **S** = **AA**^{T}
  Cholesky decomposition
  Singular Value Decomposition <- more on this later
  Matrix square root
- Whiten the data:

$$\mathbf{x} \leftarrow \mathbf{A}^{-1}(\mathbf{x} - \mathbf{m})$$

68

Now we can reverse the process to make out data look like it came form a standard MVN

Compare this to the standardisation operation: there, we subtract the mean, and multiply buy the inverse of the standard deviation. Here we do the same, but in multiple dimensions (note that the standard deviation is the square root of the variance, just like the A matrix squared is the *co*variance).

---

dealing with too many features

Feature selection:

Select a *subset* of the existing feature to operate on.

Dimensionality reduction:

Map the features to a new *smaller* set of features. Retain as much information as possible.

for now: only unsupervised methods.

69

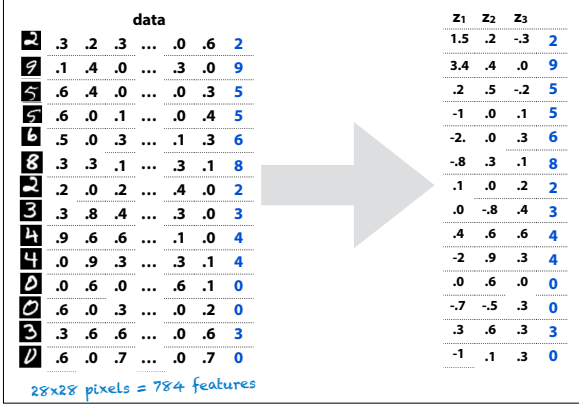
Some datasets have more features than a given model can handle. In that case, there are two things we can do: we can try to find a subset of the features that is most informative, and operate on those. This has the benefit that the features retain their meaning and are still interpretable. This is called **feature selection**.

Dimensionality reduction takes information from *all* features and maps them to a new (smaller) set of derived features, which retain as much of the original information as possible. In this case, the new features don't always have an obvious meaning, but they may still work well for machine learning purposes.

For now, we will look at one dimensionality reduction method: Principal Component Analysis (PCA).

---

dimensionality reduction



70

Dimensionality reduction is the opposite of the feature expansion trick we saw earlier. It describes methods that reduce the number of features in our data (the dimension) by deriving new features from the old ones, hopefully in such a way that we capture all the essential information. There are several reasons you might want to reduce the dimensionality of your data:

- Efficiency. Many machine learning methods can only handle so many features. If you have a very high dimensional dataset, you may be forced to do some dimensionality reduction in order to be able to run your chosen model.
- Reduce variance of the model performance (make the bias/variance tradeoff). Feature expansion boosts the power of your model, likely giving it higher variance and lower bias. Dimensionality reduction does the opposite: it reduces the power of your model likely giving you higher bias and lower variance.
- Visualization. If you're lucky (or if you have a very strong dim. reduction method), reducing down to just 2 or 3 dimensions preserves the important information in your data. If so, you can do a scatterplot, and use the power of your visual cortex to analyse your data (i.e. you can look at it).

reduction to 1 dimension

assumptions:

data is mean-centered
i.e. we have already subtracted the mean from the data

linear transformation
the same transformation is applied to all instances

$$x_1 \approx zc_1, \ldots, x_m \approx zc_m$$

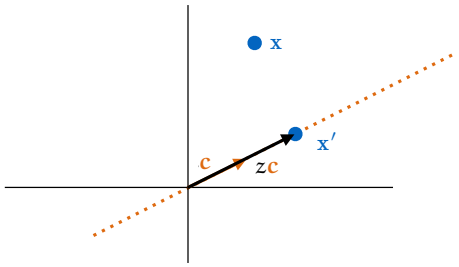
$$\mathbf{x} \approx \mathbf{x}' = z \cdot \mathbf{c}$$

71

We will start by illustrating the principle for reducing a dataset to one dimension. We will make the assumption that the transformation to the new single feature is linear.

So, our new feature z is the dot product of the data with a vector **c**, which we are going to learn.

$$\mathbf{x}' = z\mathbf{c}$$



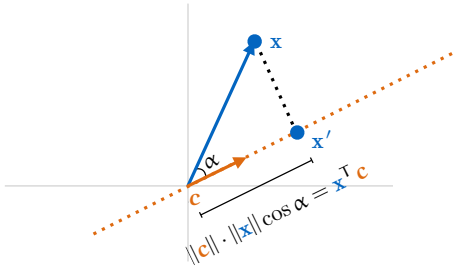
72

Here what that looks like on two dimensions.

Given some **c**, what should z be?

The closest we can get **x'** to **x** is to put **x'** where the line between **x** and **x'** makes a right angle with the line of **c**. This is the projection of **x** onto **c**, and if you know your linear algebra, you'll know the length of z**c** is related to the dot product of **x** and **c**. Why?

$$\mathbf{x}' = z\mathbf{c}$$

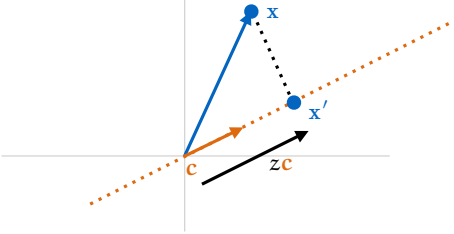


assume that **c** is a unit vector

73

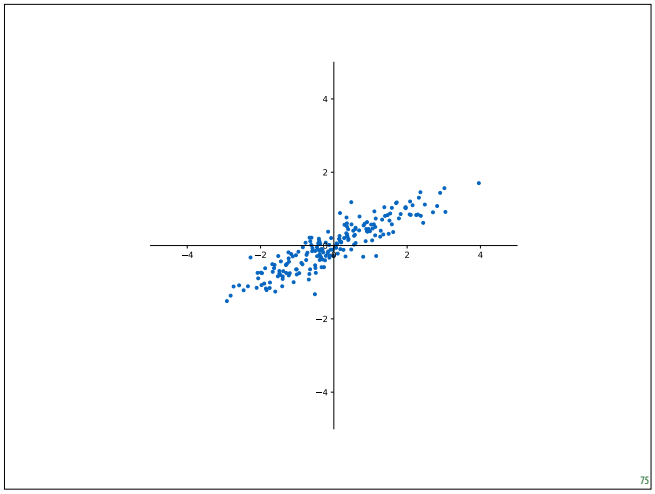
For our purposes, the length of **c** doesn't matter, so we'll assume that it has length 1 (that is, it is a **unit vector**).

From basic trigonometry, we know that the length of the black line is ||**x**|| cos α. Because ||**c**|| = 1, we can multiply by that without changing the value, which means that the length of the black line is equal to the dot product between **x** and **c**.

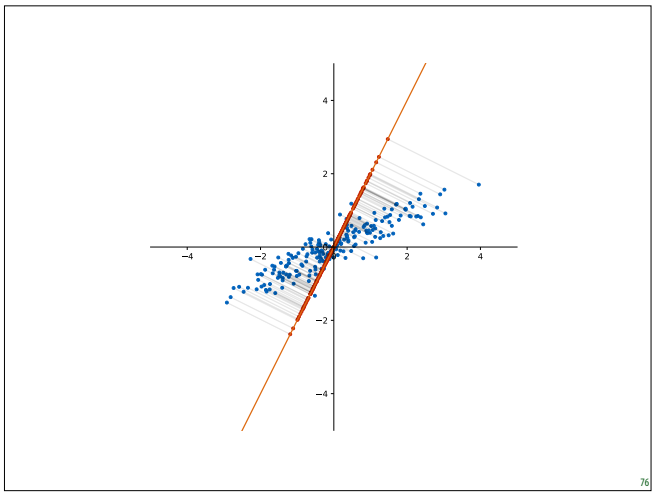


$$z = \mathbf{x}^T\mathbf{c}$$

Since **c** has length 1, the optimal value of z is just the dot product between **c** and **x**.



Here is some data. We'll pick an arbitrary **c**, and check what our data looks like with reduced dimensionality.



Here it is. The red points are our reconstructions. For each point, the new feature z is the distance from the origin to the red point. The grey lines indicate how far the reconstruction is from the original data.

Clearly, this is not a very good choice for **c**. The gray lines could be much shorter. We can think of optimizing **c** as making the grey lines rubber bands, that pull on the line representing **c** (which pivots around the origin).

---

objective

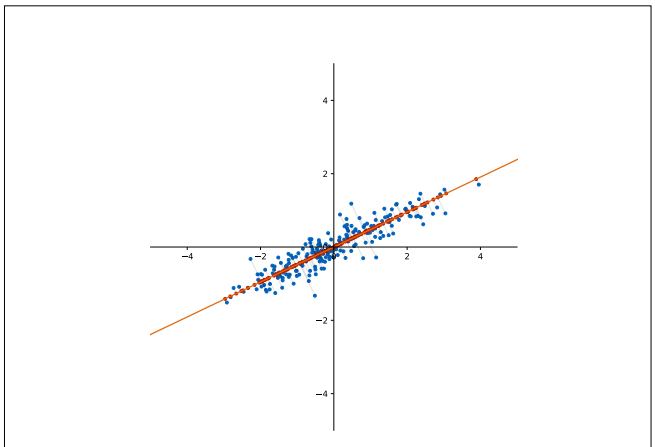
$$
\begin{aligned}
&\arg\min_{\mathbf{c}} \|\mathbf{x}' - \mathbf{x}\| \\
&= \arg\min_{\mathbf{c}} \|z \cdot \mathbf{c} - \mathbf{x}\| \\
&= \arg\min_{\mathbf{c}} \|\mathbf{x}^T\mathbf{c} \cdot \mathbf{c} - \mathbf{x}\| \\
&= \arg\min_{\mathbf{c}} \sqrt{\sum_i \left(\mathbf{x}^T\mathbf{c} \cdot c_i - x_i\right)^2} \\
&= \arg\min_{\mathbf{c}} \sum_i \left(\mathbf{x}^T\mathbf{c} \cdot c_i - x_i\right)^2
\end{aligned}
$$

such that $\|\mathbf{c}\| = 1$

77

We haven't covered optimization under constraints yet, but just imagine doing gradient descent with a little extra trick. For this objective it doesn't make much of a different, since we know there is an optimal solution for every length of c. We're just choosing the one that has ||**c**||=1.

To implement this I just to the square of ||c||-1 (the extent to which the contraint is violated) and added that to the loss function as an extra error. That doesn't normally guarantee that the solution satisfies the constraint, but in this case it does.



This is the optimal solution.

for more dimensions

repeat the process on the remaining directions.

the second principal component c_2 is the unit vector

- orthogonal to c_2
- which, together with c_2, minimises the loss

79

The third principal component is orthogonal to the first two, and so on.

another perspective

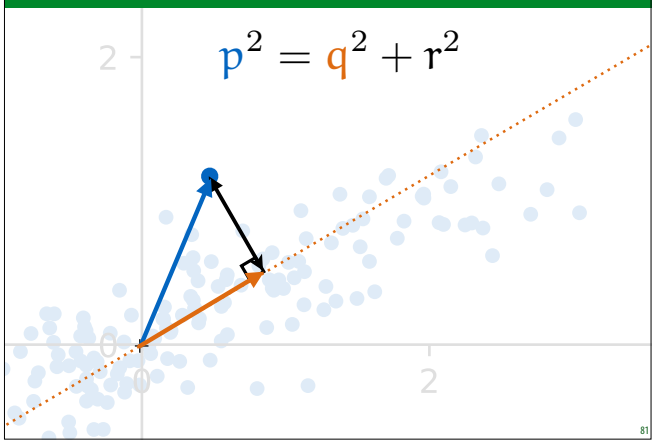
The first principal component is the direction in which the *variance* is maximal.

The second principal component is the direction in which the remainder of the variance is maximal.

And so on.

80

least squares *is variance maximisation*



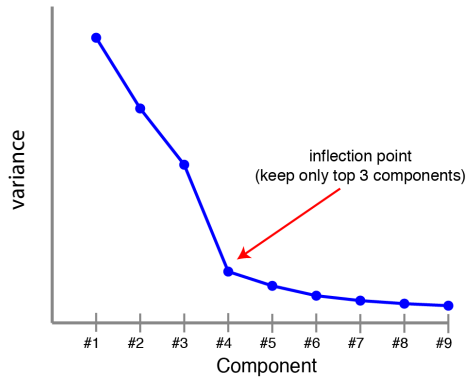
81

The variance of a dataset is defined as the mean of the squares of all the distances to the mean.Both the data and the reconstruction are mean-centered, so we are looking at the squares of the orange arrow for the variance of the reconstructions. Maximising the variance means maximizing the square of the size of the orange arrows (q).

The arrangement into a right-angled triangle means that the variance of the original data (p, the squared distance to the mean) is related to the variance of the projected data (q)and the reconstruction error (r, in black) by the Pythagorean theorem.

Since p, the variance of the original data, is a constant, $q^2 + r^2$ is constant, and minimizing the term r^2 is equivalent to maximising the other term q^2.

choosing the nr. of principal components

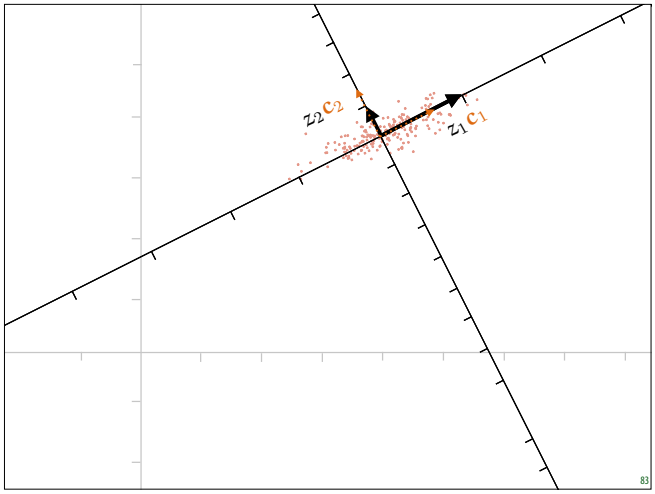


source: http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/

82

To choose the number of components, it's helpful to plot the objective (either variance or reconstruction loss) against the number of components and look for an *inflection* point.

What if we keep going until the number of components is equal to the original number of features?



If we do that, we still get a rebasing of the data. It turns out, that this gives us a whitening of the data. If we take the **c** vectors as an orthonormal basis, and then stretch by the corresponding z values, we get a coordinate system in which the new data looks like a standard normal distribution (or as much like a standard normal distribution as we can achieve with a linear transformation).

yet another perspective

PCA *whitening*:

- apply PCA with k=m
- Use $z_i \mathbf{c}_i$ as basis vectors

84

Note that while z and **c** together is not an orthonormal basis, c by itself is. Thus, we can still easily transform back and forth between the whitened basis and the original data coordinates.

but wait…

Doesn't PCA have something to do with eigenvectors?

Only if you want to compute it *efficiently*.
Or understand it even better

week 6: matrix models

85

Optimizing one dimension at a time is a terrible way to actually compute the PCA. We will look at a good way to compute it (the eigenvalue and singular value decompositions) in week 6.

PCA

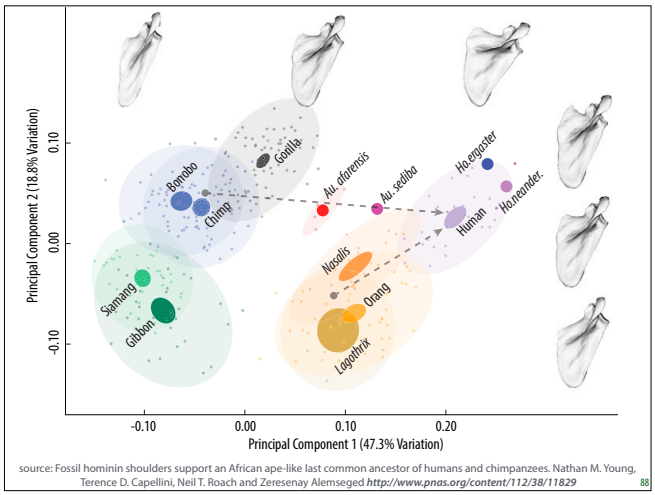
Principal Component Analysis

- linear transformation that minimises MSE loss
- linear transformation that maximises variance
- whitening transformation
- orders dimensions by variance captured/rec loss
  The first dimension is the most important for reconstructing the data, then the second and so on.

86

So what's the point?

87



source: Fossil hominin shoulders support an African ape-like last common ancestor of humans and chimpanzees. Nathan M. Young, Terence D. Capellini, Neil T. Roach and Zeresenay Alemseged *http://www.pnas.org/content/112/38/11829*

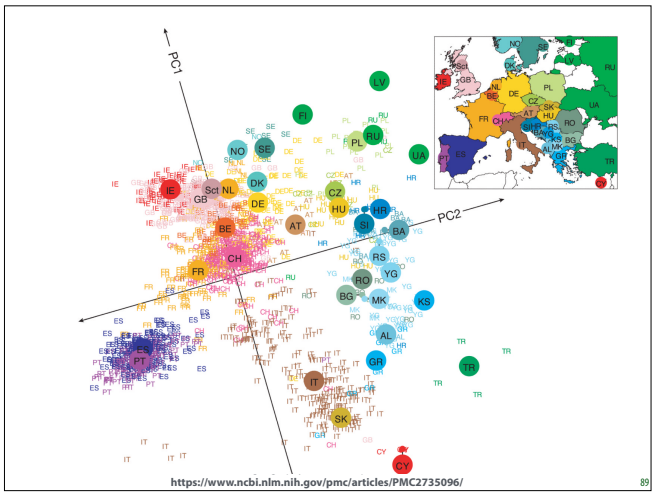
88

Here is an example of how PCA is used in research. An anatomist specialising in primates can easily tell for a single bone in isolation that it's an early hominid fossil (very rare) and not a chimpanzee fossil (not rare). But how to substantiate this? "It's true because I can see that it is" is not very scientific.

Here's one common approach. Take a large collection of the same specific bone (the *scapula,* or shoulder blade, in this case) from different apes and humans, and take a bunch of measurements (features) of each. Do a PCA, and plot the first two principal components. As you can see, the different species form very clear clusters, even in just two dimension.

We can now show that a new fossil we've found is clearly closer to human than to chimp just by measuring it, and projecting it into this space.

source: Fossil hominin shoulders support an African ape-like last common ancestor of humans and chimpanzees. Nathan M. Young, Terence D. Capellini, Neil T. Roach and Zeresenay Alemseged ***http://www.pnas.org/content/112/38/11829***



89

Here is another example. In this research, the authors took a database of 3000 Europeans and extracted features from their DNA. They used about half a million sites on the DNA sequence where DNA varies among humans (i.e. 3000 instances: people, and 500k features: DNA markers).

The two principal components of this data largely express how far north the person lives, and how far east the person lives, which means that the large scale geography of Europe can be extracted from our DNA. If I sent a large sample of European DNA to some aliens on the other side of the galaxy who'd never seen our planet, they could use it to get a rough idea of our geography.

eigenfaces

```
from sklearn import datasets

faces = datasets.fetch_olivetti_faces()
```

90

Finally, possibly the most magical illustration of PCA: **eigenfaces**. Here we have a dataset (which you can easily get from sklearn) containing 400 images, in 64x64 grayscale, of a number of people. The lighting is nicely uniform and the facial features are always in approximately the same place.

We take each pixel as a feature, giving us 400 instances each represented by a 4096-dimensional feature vector.

We'll see where the eigen- prefix comes from in week six.

---

mean face

91

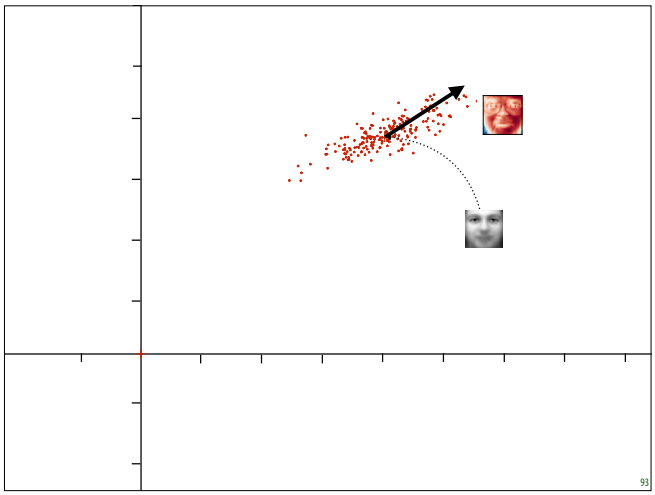
Here is the sample mean of our data, re-arranged back into an image.

---

components

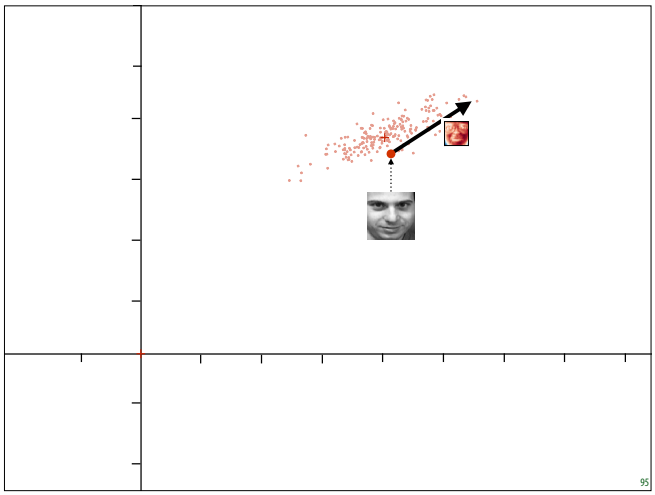
92

These are the first 30 eigenvectors (top left is the first, to the right of that is the second and so on). We've re-arranged them into images, but these are just vectors in our 4096-dimensional space. They are the basis vectors that are most natural for our data.



93

Starting from the mean face (in the middle column), we can take little steps along the direction of one of our eigenvectors (or in the opposite direction). We see that moving along the first eigenvector roughly corresponds to ageing the face. Moving along the fourth seems to make the face more female.



Here is the same, but starting at one of the images of the dataset. The middle column represents the starting point. To the right we add the k-th principal component, to the left we subtract it. Note the effect of the fifth eigenvector: subtracting it opens the mouth, and adding it seems to push the lips closer together.

If **e** is the vector describing the image in the new coordinates found by PCA, we can reconstruct the image by starting with the mean (top left), adding e_1 times the first eigenvector, adding e_2 times the second to that and so on. Since the eigenvectors are arranged by variance, the first has the highest impact. As we add more and more eigenvectors, we get gradually closer to the original image.

At bottom right is the reconstruction after 60 eigenvectors.

---

mlcourse@peterbloem.nl
